①

UMIACS-TR-90-27 February 1990
CS-TR -2416

Using Integrity Constraints to Control Search in Knowledge Base Systems

Theresa Gaasterland*, Mark Giuliano*, Anne Litcher*, Yuan Liu*, and Jack Minker*,†

Institute for Advanced Computer Studies†
and
Department of Computer Science*
University of Maryland, College Park, MD 20742

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-90-27 February 1990
CS-TR -2416

Using Integrity Constraints to Control Search in Knowledge Base Systems

Theresa Gaasterland*, Mark Giuliano*, Anne Litcher*, Yuan Liu*, and Jack Minker*,†

Institute for Advanced Computer Studies†
and
Department of Computer Science*
University of Maryland, College Park, MD 20742

Abstract

A knowledge base system combines data management and reasoning capabilities. It can be used to store and manipulate a body of knowledge which consists of a set of rules and facts. The rules and facts in a knowledge base capture syntactic information. In contrast, integrity constraints contribute semantic information about the represented knowledge. They impose restrictions on the states of the world that a knowledge base can model. Typically, integrity constraints are used to update and maintain the knowledge base, but they can also be a powerful tool for answering queries.

A logic programming system augmented with constraint processing, data storage, and data manipulation capabilities forms the basis for a knowledge base system. Both a runtime approach and a compiled approach to using inegrity constraints in logic programming systems to identify and eliminate unproductive search activity have been implemented within an existing parallel logic programming system, PRISM. We have thus extended PRISM to be a testbed for knowledge base applications. The extended system provides the basis for a series of experiments which 1) compare the performance of the compiled approach and the runtime approach, 2) demonstrate that significant classes of knowledge representation domains can use integrity constraints effectively, and 3) reveal additional techniques necessary to put the theoretical approaches into practice. We show that using constraints to process a query can reduce search space and response time. Furthermore, we show that the compiled approach reduces response time much more than the runtime approach. (KR)

1 Introduction

A knowledge base system combines data management and reasoning capabilities. It can be used to store and manipulate a body of knowledge which consists of a set of rules and facts. Tools exist to build such systems: a database system stores and manipulates sets of facts; a logic programming system has deductive capability - given a a query and a set of rules, possibly containing function symbols, and facts, it provides an answer. Additional knowledge base services, which can be implemented as metaprograms in logic program sytems, include the ability to explain answers, to update the rules and facts, to perform query optimization, and to ensure the security of sensitive information.

The rules and facts in a knowlege base capture syntactic information. In contrast, integrity constraints contribute semantic information about the represented knowledge. They impose restrictions on the states of the world that a knowledge base can model. Typically, integrity constraints are used to update and maintain the knowledge base, but they can also be a powerful tool for answering queries. Although they add no deductive knowledge - all answers deducible with the use of integrity constraints can be deduced without the use of integrity constraints - they provide information about the facts and rules that can be used to control the deductive process.

Specifically, integrity constraints allow the description of impossible states, necessary states, and dependencies between knowledge base predicates. For example, a knowledge base concerning genealogy might have an associated integrity constraint which says that no one can be both the father and the mother of somebody. If the knowledge base satisfies this constraint, then a query asking who is both a mother and a father will certainly fail. A query answering process which considers constraint information while seeking an answer to the query could answer the query with a partial deductive search - in some cases with no search.

A logic programming system augmented with constraint processing, data storage, and data manipulation capabilities forms the basis for a knowledge base system. [7] and [2] provide formal frameworks for using integrity constraints in logic programming systems to identify and eliminate unproductive search activity. A runtime approach [9] involves processing a query simultaneously on separate processors, a deductive processor and a constraint processor. The deductive processor searches for an answer in a traditional manner while the constraint processor checks to see if any intermediate state of the deduction violates a constraint or is subject to a constraint. A compiled approach [1] transforms the rules to incorporate the constraints into the rule bodies. As the transformed rules are used in a deductive process, checks can be performed on the bindings of variables to ensure that they will not lead to impossible or disallowed states.

Existing logic programming systems generally do not handle integrity constraint information during the problem solving portion of responding to a query. For example, consider a constraint of the form $\leftarrow L_1 \ldots L_n$ where each L_i is a literal and the conjunction of literals is an impossible state. Prolog would take such a constraint and process it as a query without gaining productivity from the activity. However, as shown in [7], [1], and [11] integrity constraints can be used in query processing by treating them as special formulas distinct from queries. As observed in [5], this treatment extends to knowledge base systems.

To evaluate how useful constraints can be in practice, we have implemented the compiled and the runtime constraint handling approaches within the parallel logic programming system, PRISM [6]. We have thus extended PRISM to be a testbed for knowledge base applications. The extended system provides the basis for a series of experiments which 1) compare the performance of the compiled approach and the runtime approach, 2) demonstrate that significant classes of knowledge representation domains can use integrity constraints effectively, and 3) reveal additional techniques necessary to put the theoretical approaches into practice.

This paper describes our system for utilizing constraints, shows how to use constraints effectively in three classes of knowledge domains, and evaluates the relative performance of the two constraint handling methods. Section 2 formally defines integrity constraints and describes the compiled and runtime approaches to optimizing query processing. Section 3 provides a description of PRISM and describes how a separate parallel processor handles integrity constraints for PRISM in the runtime approach. Section 4 characterizes knowledge domains in which constraints are particularly useful and evaluates the two approaches using the results obtained by running example programs. The final sections identify future work and summarize our findings. Some familiarity with Prolog or another Horn clause programming language will help the reader in understanding this paper.

2 Integrity Constraints

Both the runtime approach and the compiled approach to handling integrity constraints are based on two techniques called partial subsumption and variable substitution. In this section, we define integrity constraints and the techniques, and describe each constraint handling approach. This information will be used in Section 3 to describe the implementation of the runtime approach and in Section 4 to describe the experiments which were performed to evaluate the two approaches.

2.1 Definitions

A *knowledge base system* has two components which are the knowledge base itself and a set of metaprograms which manipulate the information in the knowledge base: KBS=<K,MP>.

A *knowledge base* has two components, the theory TH, and the set of integrity constraints IC: K=<TH,IC>. We consider TH to consist of two distinct sets of clauses, the intensional database and the extensional database: TH=<IDB,EDB>.

The *intensional database* (IDB) is a set of clauses of the form $A \leftarrow B_1, \ldots, B_n$. where A and each B_i is an atom. A is called the *head* of the clause, and $B_1, \ldots, B_n$ is called the *body*. All the variables in a clause are assumed to be universally quantified.

The *extensional database* (EDB) is a set of clauses of the form $A \leftarrow$. EDB clauses are also called facts, and IDB clauses are also called rules.

The theory component of a knowledge base, <IDB,EDB>, can also be considered a *logic pro-*

gram.

A *query* is a clause of the form $\leftarrow B_1, \ldots, B_n$. Answers to a query are obtained by expanding the query using resolution to produce a search tree. Each node of the search tree contains a conjunction of atoms that is called a *goal*. The goal in the root node is the conjunction of atoms in the query, and the goal in each node of the tree is a subgoal of the goal in the root node.

The *integrity constraints* considered in this paper are a set (IC) of clauses of the form $\leftarrow C_1, \ldots, C_n, E_1, \ldots, E_m$, where each C_i is an atom whose predicate appears in a fact or the head of a rule in the logic program, and each E_i is an atom with the evaluable predicate EQ, NEQ, LESS, or LEQ. (These evaluable predicates stand for the binary predicates $=$, $\neq$ $<$, and $\leq$, with their usual interpretations.) Each constraint may be read as "C_1 *and*...*and* C_n *and* E_1 *and*...*and* E_m can not occur".

Assuming that the theory of a knowledge base is consistent with its integrity constraints, the integrity constraints can be used to process queries by incorporating the constraints into the IDB rules to produce a semantically equivalent theory. Queries given to the new theory have the same set of answers as if they were given to the original theory. A method called *partial subsumption* to achieve this transformation was developed by [1]. Partial subsumption is the basis for both the compiled approach and the runtime approach to handling integrity constraints. Section 2.2 describes partial subsumption, and Sections 2.3 and 2.4 describe the two approaches.

2.2 Partial Subsumption

Partial Subsumption is a procedure that determines if an integrity constraint is relevant to a conjunction of literals. Let I be a constraint, and let F be a conjunction of literals.

An integrity constraint, $\leftarrow C_1, \ldots, C_n$, *partially subsumes* a conjunction of literals, $B_1, B_2, \ldots, B_m$, if and only if there exists a non-empty subset S of $\{C_1, \ldots, C_n\}$ and a substitution θ such that $S\theta \subseteq \{B_1, \ldots, B_m\}$. If $C_{i_1}, \ldots, C_{i_k}$ are the literals in the constraint that are not in S, then the clause $(\leftarrow C_{i_1}\theta, \ldots, C_{i_k}\theta)$ is called a *partial constraint.*

If I partially subsumes F and the resulting partial constraint is empty or can be evaluated to *false* – i.e. all the literals in the body of the partial constraint evaluate to *true* – then F represents a disallowed state that violates the integrity constraint. For example, suppose that *dinosaur("fred")* is a (trivial) conjunction of literals representing a state where fred is a dinosaur, and suppose that $\leftarrow$ *dinosaur(y)* is an integrity constraint that says "there are no dinosaurs". The constraint partially subsumes the conjunction with the substitution $\{y/\text{"fred"}\}$ and leaves an empty partial constraint; this tells us that Fred can not be a dinosaur. If the constraint were $\leftarrow dinosaur(y), y = \text{"fred"}$, meaning that fred is not a dinosaur, then the constraint would partially subsume the conjunction *dinosaur("fred")* leaving the partial constraint $(\leftarrow \text{"fred"} = \text{"fred"})$. The partial constraint evaluates to *false*; therefore, fred can not be a dinosaur.

If a partial constraint that does not evaluate to *false* is left after partial subsumption, then adding the partial constraint to the original conjunction of literals produces a new semantically equivalent conjunction of literals. The new conjunction of literals is said to be the product of *merging* the original constraint with the original conjunction of literals.

An integrity constraint might contain constants or multiple occurrances of a variable that prevent it from partially subsuming a conjunction of literals. In this case, the constraint can be rewritten into an equivalent *variable substituted* form that will do so [1] Each occurrance of a constant is replaced by a unique variable and an equality literal is added to the constraint which equates the new variable and the constant. Similarly, for each duplicate occurrence of a variable, the variable is replaced with a unique variable and an equality literal is added to equate the new variable with the old variable. The variable substitution algorithm appears in Appendix ??.

For example, the constraint $\leftarrow dinosaur(\text{"}fred\text{"})$ can be rewritten as $\leftarrow dinosaur(x), x = \text{"}fred\text{"}$. This variable substituted form would partially subsume the conjunction of literals $dinosaur(y)$, leaving the partial constraint ($\leftarrow y = \text{"}fred\text{"}$) which does not evaluate to *false* but rather restricts y from ever taking the value "*fred*". The partial constraint would be added to the original conjunction by attachment, producing the new semantically equivalent conjunction $dinosaur(y), y \neq \text{"}fred\text{"}$.

2.3 Compiling Constraints

In the compiled approach to handling integrity constraints, a set of constraints is combined with the bodies of the rules in a knowledge base's theory to produce a semantically equivalent theory. The new theory is said to be *semantically compiled.*

Prior to semantically compiling a knowledge base theory, the IDB rules must be flattened into a new set of IDB rules whose bodies contain only extensional or recursive predicates. This intermediate transformation eliminates the need for deduction on non-recursive predicates in the query answering process. If the integrity constraints are defined in terms of intensional predicates, they must also be flattened. Flattening is performed according to the method defined in [15]. This method can be used on a program with direct recursion by flattening the non-recursive predicates in the recursive rules before merging the rules with the constraints.

Each flattened rule of the form $A \leftarrow B_1, \ldots, B_n$ in the new set of rules is merged with the set of integrity constraints to produce a new *semantically constrained* rule of the form $A \leftarrow B_1, \ldots, B_n, PC_1, \ldots, PC_k$ where $PC_1, \ldots, PC_k$ are the partial constraints. Merging may show that a flattened rule violates an integrity constraint. Such a rule represents a useless interaction among the rules in the unflattened program and can be discarded. Otherwise, if a flattened rule does not violate any of the constraints, then the new rule is added to the set of semantically constrained rules.

A semantic compiler for a set of IDB rules and EDB facts has been implemented as a meta-level program in Prolog [11]. The semantic compiler produces a set of rules that contains all relevant constraint information but may still need additional code transformations in order to be executable. In addition, partial constraints may be disjunctions of literals; thus, unfolding may be required to ensure that rules have the correct form. The compiler was used to prepare the semantically compiled programs for the experiments.

2.4 Checking Constraints at Runtime

The runtime approach also uses the principle of partial subsumption but does so during query execution. As a query is processed by deduction, each node in the resulting search tree is checked for a violation of the set of integrity constraints. Query processing and constraint processing are treated as separate tasks that can be undertaken by different processors.

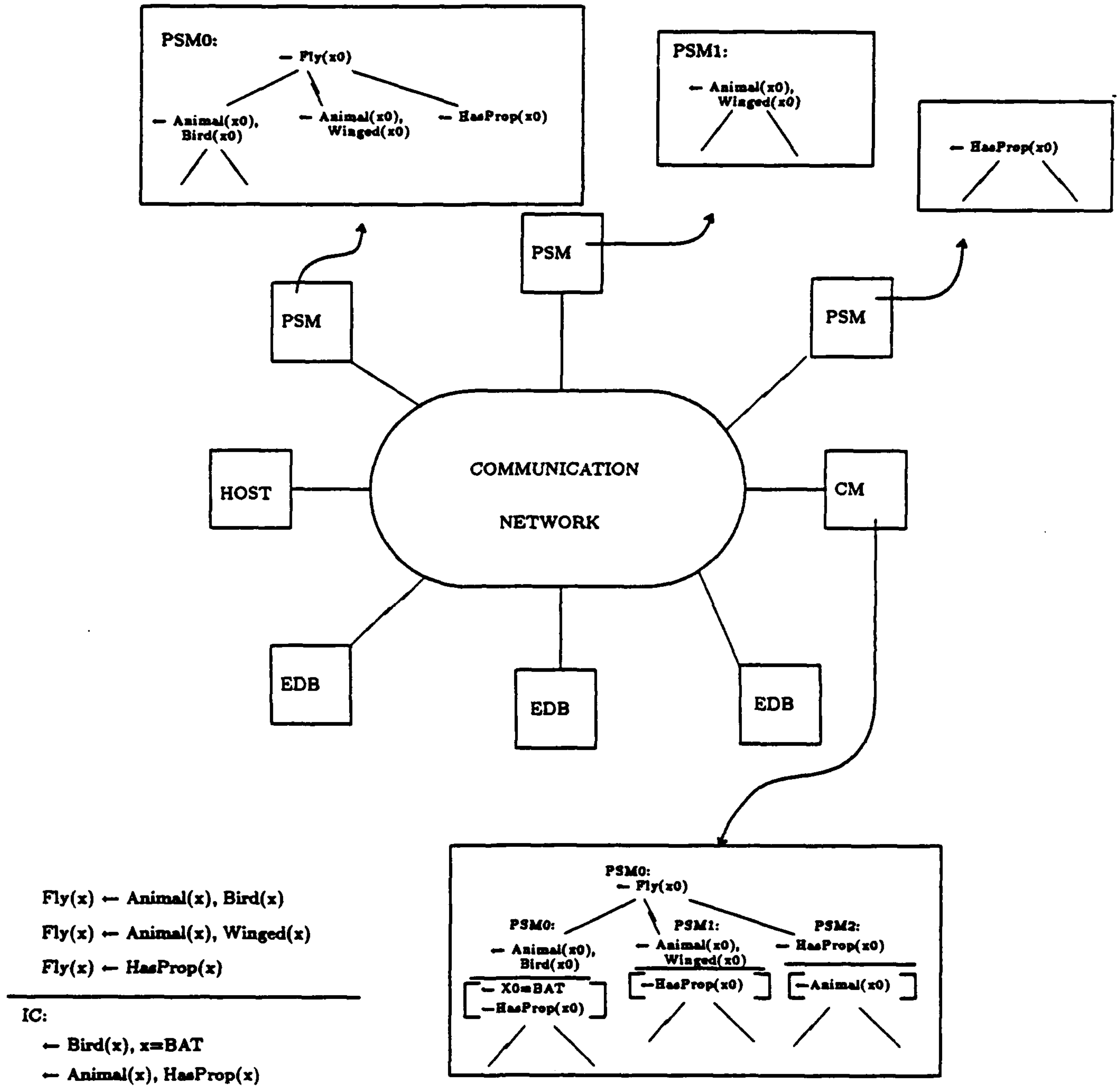
Consider a system that consists of a deductive processor and a constraint processor. The deductive processor sends nodes of a search tree to the constraint processor, and the constraint processor uses partial subsumption to detect violations in a node. The deductive processor is able to use the violation information from the constraint processor to trim the search tree below the culpable node. This is a simplified description of the way the runtime approach has been implemented in the context of PRISM. A processor called a constraint machine has been developed which performs constraint checking, creation of partial constraints, storage of nodes with partial constraints, and message passing. Section 3 describes the implementation of this runtime approach.

3 Constraint Handling in PRISM

In order to evaluate the practicality and utility of the runtime and compiled approaches to constraint handling, we incorporated each approach into an existing logic programming system, PRISM. PRISM was designed as a testbed for experimenting with alternative processor functionalities for logic programming on loosely coupled MIMD architectures. Thus, PRISM can handle deduction over the theory, or TH, component of a knowledge base system. When augmented with constraint handling capability, PRISM can also handle the IC component and becomes a testbed for knowledge base applications. A description of PRISM is given in [6].

The PRISM system executes logic programs by delegating portions of a query's implicit AND/OR tree to different processors – a dynamically configured set of problem solving machines (PSMs) which communicate by passing messages. A PSM is initialized with a conjunctive goal. Two alternative procedures may be used to expand this or any subsequent goal in the proof tree: either splitting a goal into its conjunctive components, or selecting a literal from a goal and unifying it with the heads of the procedures in the logic program. Goal expansion may result in a proof tree with multiple active branches; the problem solver may elect to distribute all but one of the active branches to alternative problem solving machines. This simulataneous expansion of multiple branches in the proof tree improves the query execution time for many applications. Although the system can exploit parallelism transparently, the user has the option of annotating programs to explicitly control execution. [4, 8, 6] provide more information on PRISM.

Incorporating the compiled constraint handling method into PRISM required adding a preliminary compilation phase to merge integrity constraints with the rules of logic programs. In contrast, to incorporate the runtime method, a new type of machine was developed which manages constraint information internally and communicates with the other PRISM machines. Figure 1 illustrates how the constraint machine fits into the PRISM architecture.



Fly(x) ← Animal(x), Bird(x)

Fly(x) ← Animal(x), Winged(x)

Fly(x) ← HasProp(x)

IC:

← Bird(x), x=BAT

← Animal(x), HasProp(x)

Figure 1: The Constraint Machine within PRISM

3.1 The Constraint Machine

The constraint machine (CM) monitors nodes in the proof tree built during query evaluation to detect whether some branch of the tree involves a conjunction of literals that violates any of the integrity constraints. A CM runs asynchronously with the other processes and communicates with PSMs through messages. Two kinds of messages are passed between a CM and a PSM: CHECK-CONSTRAINTS messages from a PSM to a CM, and VIOLATION messages from a CM to a PSM. To check for constraint violations, the CM uses partial subsumption to maintain its own version of the proof tree as it participates in ongoing communication with the PSMs.

3.1.1 Maintaining Constraint Information

The Constraint Machine's version of the proof tree for a query tree carries the information necessary to check constraints dynamically as the query is being solved. Although the structure of the CM's proof tree is the same as the structure of the PSM's proof tree, the contents of the nodes differ. While each PSM node contains a subgoal in the proof, each CM node contains the partial constraints that the corresponding PSM subgoal and its descendents must not violate. As the PSM generates each new node in its tree, it sends information about the new node to the CM so that the CM may build its corresponding node. We call the two trees the constraint tree and the search tree.

To generate a new search tree node, the PSM takes an atom from a search tree node and attempts to unify the atom with the head of some program clause. If the unification succeeds, with a unifier θ, a descendent node is produced by attaching the body of the clause used in the unification to the rest of the literals in the parent node and applying θ to all literals in the new node. Since the CM has already checked the parent node, only the differences between the descendent node and its parent need to be checked to detect any new violations. Thus, the PSM sends a CHECK-CONSTRAINTS message to the CM containing the unifier θ, the list of newly attached literals and the node numbers of the new node and its parent.

With this information, the CM can generate the partial constraints associated with the new PSM node and if no violation occurs, it can place a new node containing the partial constraints in the constraint tree. If a violation does occur, the CM sends a VIOLATION message to the PSM to indicate that the PSM should curtail search below the node in question.

3.1.2 Testing for Violations

When a CM receives a CHECK_CONSTRAINTS message with a list of new literals for a search tree node N and a substitution θ, the following algorithm is performed to either send a VIOLATION message to the PSMs, or to create a (possibly empty) list LN of partial constraints for node N:

1. Initialize LN to be the empty list. Initialize L to be the list of all the integrity constraints in IC.

2. If N is a node that is not the root of the search tree, then find the partial constraints list, L$'$, of the parent node of N. For each partial constraint C in L$'$, if any of the evaluable literals in $C\theta$ can be evaluated to *false* then do nothing (this means that the partial constraint $C\theta$ will not be violated by N or any of N's descendants in the search tree); otherwise, add $C\theta$ to L.

3. Use the Partial Subsumption Algorithm in B to test N and each member C of L. The result of the test will be one of the following:

   - If C partially subsumes the new literals in N and produces an empty partial constraint then send a VIOLATION message to the PSMs and go to the next step.
   - If C partially subsumes the new literals in N and produces no empty partial constraints then add the resulting new partial constraint(s) to LN.
   - Otherwise, C does not partially subsume the new literals in N. If C is one of the constraints in IC then do nothing (this means that C is not relevant to N). Otherwise, C is a partial constraint inherited from the parent of N; add C to LN.

4. Halt.

3.1.3 Evaluable Predicates and Equality in the Constraint Machine

To carry out the test for violations, the CM must be able to evaluate literals with evaluable predicates. An attempt to evaluate a literal has three possible results: *true*, *false*, or *delay evaluation*. Unlike the other evaluable literals, when a literal whose predicate is EQ evaluates to *true*, it returns a (possibly empty) substitution θ to be applied to the partial constraint containing the EQ literal.

Variable substitution may introduce new variables that do not participate in partial subsumption. Rather, these variables become instantiated by substitutions produced by EQ evaluation. The compiler that prepares integrity constraints for use in the CM performs variable substitution on all the integrity constraints. If an integrity constraint contains any function symbols, the variable substituted form will contain new variables that occur only in EQ literals. For example, consider the constraint $\leftarrow P(f(x,y)), Q(f(y,x))$ whose variable substituted form is $\leftarrow P(x0)$, $Q(x1)$, $EQ(x0,\ f(x,y))$, $EQ(x1, f(y,x))$. The variables x and y now occur only in the EQ literals. We call these variables, which were isolated in EQ literals by variable substitution, *EQ-only variables*.

If an EQ literal contains two terms that can be unified with a substitution θ that replaces EQ-only variables and no others, then the literal evaluates to true and θ is applied to the other literals in the partial constraint; otherwise the evaluation is delayed. In the example, all EQ literals must be delayed.

If the CM receives from the PSM the trivial list of literals $P(f(a,b))$, then it constructs a new partial constraint $\leftarrow Q(x1), EQ(f(a,b), f(x,y)), EQ(x1, f(y,x))$. The first EQ literal can be evaluated to *true* with the substitution $\{a/x, b/y\}$ to produce $\leftarrow Q(x1), EQ(x1, f(b,a))$. The variable x1 is not an EQ-only variable, so the EQ literal must be delayed. Finally, if the PSM sends the list $Q(f(c,c))$ resulting in the new partial constraint $\leftarrow EQ(f(c,c), f(b,a))$, the EQ literal evaluates to *false*.

The evaluation of EQ is defined formally as follows:

- if the arguments are ground and not syntactically identical, return *false*.
- if the arguments are syntactically identical, return *true*.
- if there exists a substitution θ that makes the arguments syntactically identical and which replaces EQ-only variables with terms containing no EQ-only variables, return *true* and apply θ to the other literals in the partial constraint.
- otherwise, delay evaluation.

4 Experiments, Results, and Evaluation

A series of experiments were performed to: 1) compare two different constraint handling methods, and 2) investigate whether significant classes of logic programs can use integrity constraints effectively to process queries. We selected three domains which characterize widely used classes of logic programming problems. The domains involve the representation of inheritance hierarchies, the use of data which tends to cluster around discrete values, and encoding generate-and-test sequences [10]. To obtain data to evaluate the utility of the constraint handling methods, three representative programs were encoded and executed in PRISM using three execution modes: without constraints, with CMs, and with constraints compiled into the program. To compare the performance of the programs run under each method, we gathered response times, machine utilization statistics, and data about communication overhead.

Our results show that using constraints improved query response time over not using constraints with a reasonably small overhead cost. The cost of the CM method is the reallocation of half of the processors from PSMs to CMs and an increase in the message traffic between machines. The major cost of the compile time method was the cost of performing the compilation.

Our results also show that for all three programs, compiling the constraints reduced response time more than using CMs did. We found that semantic compilation can affect programs in two ways: it can statically reduce search space, and it can introduce runtime tests for pruning the search space. When the partial constraints created by semantic compilation either are null or contain only evaluable literals that are sufficiently bound to be evaluated at compile time, the constraints statically reduce the search space by eliminating rules. Other types of partial constraints introduce runtime tests. The relative performance of the different configurations depends on the complexity of the new runtime tests and their potential to reduce the search space.

Semantic compilation produced programs that could not be executed without further manipulation. The compiled rules in the inheritance hierarchy program contained redundant partial constraints – that is, some partial constraints were subsumed by other partial constraints. For efficiency, the redundant constraints were omitted in the executable program. To make the constraints useful in the data clustering program, the variable substitution algorithm had to be augmented with knowledge about basic number theoretic principles. For the generate-and-test program, it was not possible to place these partial constraints in rules so that they prune search space under a left-to-right control strategy. A new control strategy was required to delay evaluation of partial constraint literals until they were sufficiently instantiated.

This section is organized as follows: Section 4.1 reviews the testing environment. Sections 4.2 and 4.3 present the results for the inheritance hierarchy domain and for the data clustering domain. Section 4.3.3 compares the results for these domains. Section 4.4 discusses our results and observations about generate and test programs. Section 4.5 compares the results for all three domains.

4.1 Testing Environment

The experiments involved running each program using a series of PRISM configurations. The configurations were determined by varying 3 parameters: the number of processors used, the tasks dedicated to each processor, and the version of the program.

A maximum of six processors were used to run programs. For the experiments using only problem solving machines (PSMs), configurations consisting of 1, 2, 4, and 6 PSMs were used to execute the programs without constraints and with constraints compiled in. For the experiments using constraint machines (CMs) as well as PSMs, each PSM was coupled with a CM. The configurations for these experiments consist of 1,2, and 3 PSM-CM pairs. In initial tests, we ran configurations with 1 CM handling all PSM nodes, but we found that in general CMs overload quickly when handling more than one PSM.

Each original program included a rule with the form Query(X) $\leftarrow P_1, ..., P_n$ which defined a representative query for the program. Queries to the program were of the form $\leftarrow$ Query(X). For each query, we used each configuration to collect the response time, the number of nodes expanded in the proof trees, and the number of messages passed. Other data was obtained from observations about the structures of the programs.

The BBN Butterfly version of PRISM was used to run the experiments. The BBN Butterfly is a hybrid shared memory machine. Each processor has fast access to its own local memory and can access memory of other processors at a slower rate. In this implementation of PRISM the shared memory of the Butterfly is used to implement a message passing system. Each PRISM processor maintains a disjoint address space and shares information with other processors through the message passing system. Each PRISM machine (a PSM or a CM) corresponds to a physical processor on the Butterfly.

Three benchmark programs were used in the evaluation, one from each of the selected domains. The first program, *animal*, determines properties of animals based upon deductive rules and a type hierarchy. Constraints are used to enforce disjoint relations in the type hierarchy. The second program, *chemistry*, determines the properties ot elements in the periodic table. Constraints are used to perform inequality checking. The third program, *zebra*, is a generate-and-test program which assigns items to houses based upon a set of properties. Constraints are used to mix the generation and testing of items. Our program is a smaller version of the *zebra* program found in the logic programming literature. [16] The source listings for all three programs appear in Appendix C.

4.2 Inheritance Hierarchies Results

Some applications, such as representing inheritance hierarchies, model data that fall naturally into *disjoint classes*; in fact, modelling these data without constraints is difficult. When disjointedness requirements are coded as integrity constraints, nodes in the proof tree which would require an element to belong to two or more disjoint classes can be identified easily and pruned.

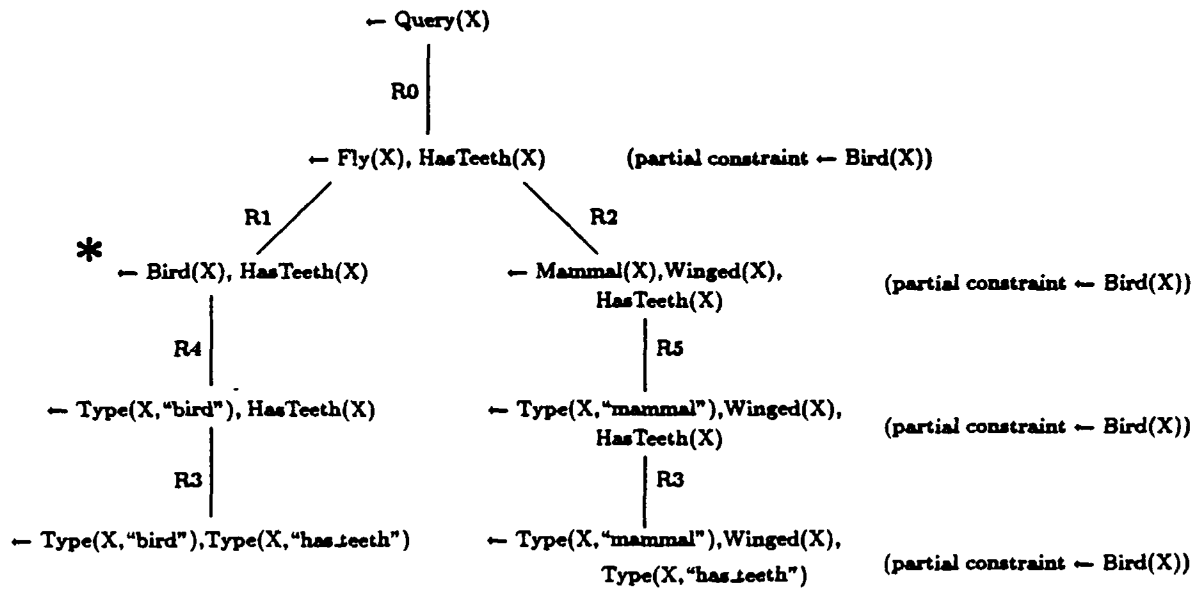
For example, consider a small type reasoning system about animals, with the following query, IDB rules and an integrity constraint that says "no birds have teeth" (i.e. Bird and HasTeeth are disjoint classes):
Q: ← Query(X)
R0: Query(X) ← Fly(X),HasTeeth(X).
R1: Fly(X) ← Bird(X).
R2: Fly(X) ← Mammal(X),Winged(X).
R3: HasTeeth(X) ← Type(X,"has_teeth").
R4: Bird(X) ← Type(X,"bird").
R5: Mammal(X) ← Type(X,"mammal").
IC: ← Bird(X),HasTeeth(X).

Using R1 to expand the predicate *Fly* in the query is unproductive, but the system will do it anyway if the query is solved without regard to constraint information. Using IC to prune the proof tree node that involves Bird(X) and HasTeeth(X) (Figure 2) eliminates the useless interaction between R1 and R3. If IC is used with the runtime approach, the subquery ← Bird(X),HasTeeth(X) is pruned after it is checked by partial subsumption and found to violate a constraint.

In the compiled method, the program would be flattened to produce:
R0′: Query2(X) ← Type(X,"bird"), Type(X,"has_teeth").
R0″: Query2(X) ← Type(X,"mammal"),Winged(X),Type(X,"has_teeth").
IC′: ← Type(X,Y1),Type(X,Y2),EQ(Y1,"bird"),EQ(Y2,"has_teeth").

IC′ partially subsumes R0′ to produce two partial constraints one of which is
{← EQ("bird","bird"),EQ("has_teeth","has_teeth"). The body of the partial constraint evaluates to *true*; thus the partial constraint evaluates to *false* and R0′ can be thrown away.

On the other hand, IC′ partially subsumes R0″ to produce the partial constraints:
{← EQ("mammal","bird"),EQ("has_teeth","has_teeth")} and
{← EQ("mammal","bird"),EQ("mammal","has_teeth")}. Each partial constraint contains an atom that evaluates to *false*; thus neither partial constraint can be violated by R0″. Thus, no partial constraint literals are added to R0″. The semantically compiled definition for the predicate Query is simply R0″.


← Query(X)
R0
← Fly(X), HasTeeth(X)
(partial constraint ← Bird(X))
R1
R2
*
← Bird(X), HasTeeth(X)
← Mammal(X),Winged(X), HasTeeth(X)
(partial constraint ← Bird(X))
R4
R5
← Type(X,"bird"), HasTeeth(X)
← Type(X,"mammal"),Winged(X), HasTeeth(X)
(partial constraint ← Bird(X))
R3
R3
← Type(X,"bird"),Type(X,"has_teeth")
← Type(X,"mammal"),Winged(X), Type(X,"has_teeth")
(partial constraint ← Bird(X))

**Pruned by CM below this node*

Figure 2: Original Search Tree for the Animal Program

Figure 2 shows the search tree for the query using the original rules and Figure 3 shows the tree for the compiled rules. The search tree for Q using R0″ is much smaller than the original tree.

4.2.1 Results and Analysis for the *Animal* Program

The *animal* program used in the experiments includes 18 integrity constraints and models a much larger set of facts about the animal world than the example above.

For the *animal* program, semantic compilation reduced the search space at compile time. The resulting partial constraints contained only evaluable literals and thus introduced no new runtime deduction. The rule for the Query predicate was flattened into 43 separate rules corresponding to 43 paths in the search tree. For all but one of the rules, the partial constraints were evaluated to *false* and the rule was thrown out. As a result, after compilation, the Query predicate was defined by a single rule in which the processing required to evaluate the partial constraint literals was negligible.

In the semantically compiled rule for Query, some of the partial constraints attached to the rule were redundant in that they were subsumed by other partial constraints. For example, the partial constraints attached to the compiled rule included {← EQ("whale",C)} and
{← EQ("whale",C),EQ("duck",B)}. The first partial constraint subsumes the second and renders

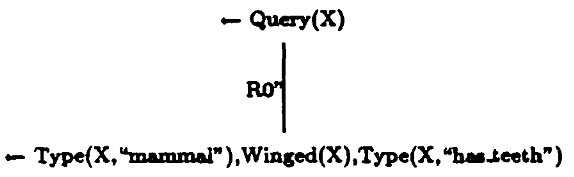


Figure 3: Search Tree for the Compiled Animal Program

the second constraint redundant. To maximize the benefit of the constraints, in a post-compilation phase we eliminated the redundant partial constraints attached to the final axiom.

As can be derived from the data in Figure 4, the compiled program's search space was 23 percent of the original search space without constraints. Running the program with CMs covered a search space that was 36 percent of the original. The CM-PSM search space was slightly larger than the compiled search space because compilation both flattens and prunes and the CM only prunes.

The response times for the semantically compiled program were much faster than for the original program. and slightly faster than for the program run with the constraint machines. Figure 5 shows the response times for the *animal* program using each configuration.

4.3 Data Clustering Results

In some knowledge domains, extensional data tend to cluster naturally according to the values taken by certain arguments. Constraints can specify the nature of the clusters and thus partition the search space. In addition, we can write rules that reflect the clustering. When a query involves a particular cluster or set of clusters, the constraints can be used to minimize the amount of data retrieval that must be done to answer the query.

For example, consider a chemistry program which captures physical properties of elements, and a query that asks for metals with melting points that are less than 961. The following is the query and a fragment of the program:

Q: ← Query(X).
R0: Query(X) ← In_Group(G,X), Is_Metal(X), Melting_Point(M,X), LESS(M,961).
R1: In_Group("IA",X) ← Member(N,[1, 3, 11, 19, 37, 55, 87]), Atomic_Number(N,X),
Is_Element(X).
R2: In_Group("IB",X) ← Member(N,[29, 47, 79]), Atomic_Number(N,X), Is_Element(X).

Method	Configuration	Number of Nodes (1)	Number of Msgs.	Size of Msgs. (2)	CM Msgs. (3)	CM Msg. Time (4)
Compiled Programs	1 PSM	68	6	115	NA	NA
	2 PSMs	68	35	1105		
	4 PSMs	68	50	1384		
	6 PSMs	68	48	1191		
With CMs	1CM 1PSM	104	114	3298	107	11.24 %
	2CMs 2PSMs	104	169	4013	122	12.34 %
	3CMs 3PSMs	104	142	3626	117	12.94 %
No Constraints	1 PSM	290	6	41	NA	NA
	2 PSMs	290	119	1660		
	4 PSMs	290	162	2180		
	6 PSMs	290	211	2376		

"msg." abbreviates "message".

(1) Number of nodes in the proof tree created while finding all answers to the query.

(2) Total size of all messages sent, in bytes.

(3) Number of messages sent by or sent to CMs.

(4) Time spent by PSMs in processing CM messages, as a percent of total PSM active time.

Figure 4: Data Collected for the Animals Program

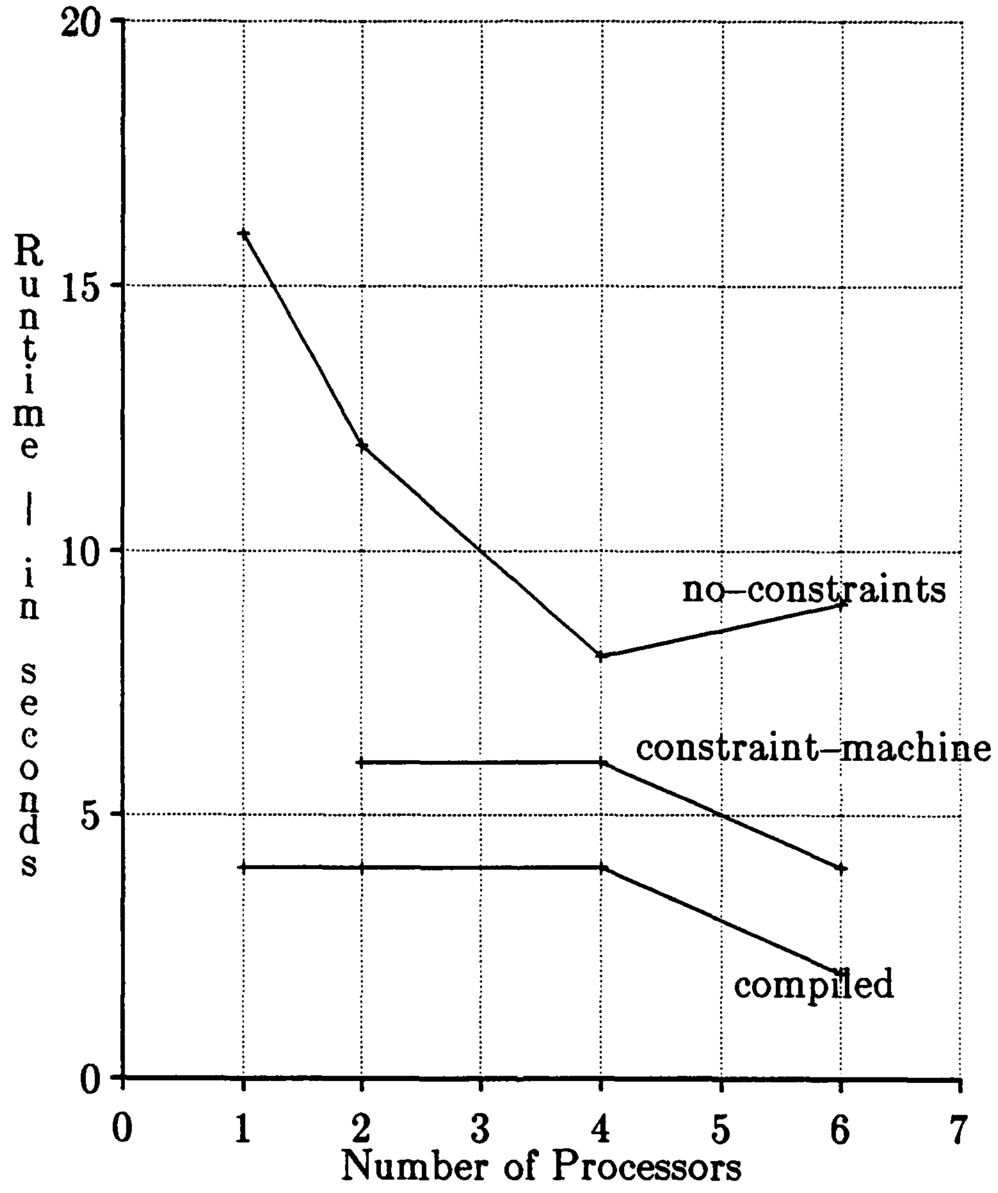


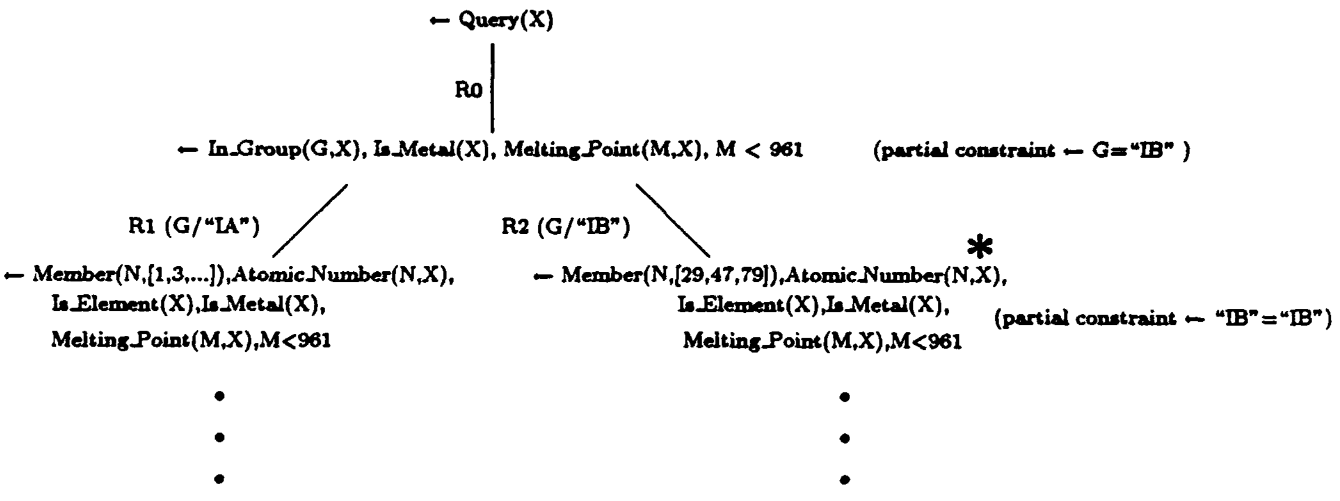
Figure 5: Runtime for the Animals Program

R3: Is_Metal(X) ← Atomic_Number(N,X), Range(N,[[2,5], [10,14], [18,32], [36,51], [54,84], [86,104]]).

The melting points of metals tend to cluster within groups in the periodic table. We can take advantage of this fact with the following integrity constraint:

IC1: ← In_Group("IB",X), Is_Metal(X), Melting_Point(M,X), LESS(M,961).

The constraint says that the melting point of any metal in group IB is at least 961, which is outside the range specified in the query. If IC1 is used to process the query at runtime, we can avoid retrieving the group IB elements by pruning the proof tree node that results when rule R2 is used to expand the atom In_Group(G,X). Figure 6 shows where the tree may be pruned.



← Query(X)
R0
← In_Group(G,X), Is_Metal(X), Melting_Point(M,X), M < 961
(partial constraint ← G="IB")
R1 (G/"IA")
R2 (G/"IB")
*
← Member(N,[1,3,...]),Atomic_Number(N,X), Is_Element(X),Is_Metal(X), Melting_Point(M,X),M<961
← Member(N,[29,47,79]),Atomic_Number(N,X), Is_Element(X),Is_Metal(X), Melting_Point(M,X),M<961
(partial constraint ← "IB"="IB")

**Pruned by CM below this node*

Figure 6: Original Search Tree for the Chemistry Program

If IC1 is semantically compiled into the query rule in the program fragment, we first get the flattened rules and flattened constraint:

R0′: Query(X) ← Member(N,[1, 3, 11, 19, 37, 55, 87]), Atomic_Number(N,X), Is_Element(X),
Atomic_Number(N1,X), Range(N1,[[2,5], [10,14], [18,32], [36,51], [54,84], [86,104]]),
Melting_Point(M,X), LESS(M,961).

R0″: Query(X) ← Member(N,[29, 47, 79]), Atomic_number(N,X), Is_Element(X),
Atomic_Number(N1,X), Range(N1,[[2,5], [10,14], [18,32], [36,51], [54,84], [86,104]]),
Melting_Point(M,X), LESS(M,961).

IC1′: ← Member(N,[29, 47, 79]), Atomic_Number(N,X), Is_Element(X),
Atomic_Number(N1,X), Range(N1,[[2,5], [10,14], [18,32], [36,51], [54,84], [86,104]]),
Melting_Point(M,X), LESS(M,961).

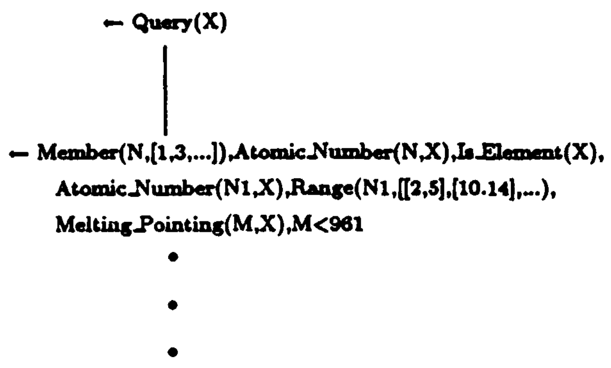

← Query(X)
← Member(N,[1,3,...]),Atomic_Number(N,X),Is_Element(X),
Atomic_Number(N1,X),Range(N1,[[2,5],[10.14],...),
Melting_Pointing(M,X),M<961
•
•
•

Figure 7: Search Tree for the Compiled Chemistry Program

Variable substitution and partial subsumption eliminate R0″ to produce a new program fragment consisting of the single rule, R0′. Figure 7 shows the search tree for the compiled program. The search space for R0′ alone is half of that for R0′ and R0″ combined.

4.3.1 Evaluable Predicates in Integrity Constraints

In the example above, the query and the integrity constraint both specify melting points that are less than 961. In contrast to this, the *chemistry* program used in the experiments contains a query rule and integrity constraints that specify distinct value ranges for a particular property. The experiments show that in order to use the integrity constraints to improve query processing, basic number theoretic principles must be used to rewrite atoms that contain the predicate LESS.

For example, suppose that a program contains the following rule:

R: Query(X) ← p(X), LESS(X,9).

The rule defines a query that asks for all the elements in relation p whose values are less than 9. Consider the following integrity constraint that says that the value of each element in relation p is at least 10:

IC: ← p(W), LESS(W,10).

The variable substitution algorithm does not replace terms in atoms containing evaluable predicates; hence the constraint IC is in variable substituted form. Applying partial subsumption to IC and R produces the partial constraint ← LESS(X,10) which can not be used to prune the subquery ← p(X), LESS(X,9) from a proof tree. In order to prune the subquery, the partial subsumption algorithm would need to use basic number theoretic principles and the atom LESS(X,9) in the subquery to figure out that if a number X is less than 9 then it is also less than 10.

In order to use integrity constraints for inequality checking in the experiments, the variable substitution algorithm was augmented with a basic number theoretic principle. Each occurrance of a constant in an atom whose predicate is LESS is replaced by a unique variable. Then, an atom whose predicate is LEQ is added to the constraint to relate the new variable and the constant (LEQ stands for less-than-or-equal). Similarly, each duplicate occurrance of a variable in an atom whose predicate is LESS is replaced by a unique variable and an atom whose predicate is LEQ is added to the constraint to relate the new and old variables. The augmented variable substitution algorithm produces the following variable substituted form for the constraint IC:

IC′: ← p(W), LESS(Y,Z), LEQ(W,Y), LEQ(Z,10).

Applying partial subsumption to IC′ and R produces the partial constraint ← LEQ(X,X), LEQ(9,10). This partial constraint evaluates to *false*; hence it can be used to prune the subquery ← p(X), LESS(X,9) from a proof tree.

4.3.2 Results and Analysis for the Chemistry Program

The experimental results for the *chemistry* program were very similar to the results for the *animal* program. In both programs, semantic compilation eliminated branches from the search space at compile time without introducing new deduction at runtime. Flattening the rule for *chemistry*'s Query predicate produced two rules. For one of the rules, the partial constraint evaluated to *false* leaving a single rule with a smaller search space.

Also like the *animal* program, both the CM-PSM search space and the compiled search space are smaller than the search space for the original program. Furthermore, the compiled search space is smaller than the CM-PSM search space because compilation cuts branches from the flattened program and the CM cuts branches from the unflattened program. As the data in Figure 8 indicate, the compiled search space was 18 percent of the original search space, and using CMs yielded a search space that was 43.5 percent of the original.

Figure 9 shows the response times for the *chemistry* program using each configuration. The compiled *chemistry* program running on one machine ran faster than the original program run on either 6 PSMs or 3 CM-PSM pairs. The response times for the original version run on PSMs-only were the slowest. Hence, the compiled version running on a sequential processor outperformed the other approaches running on up to 6 parallel processors.

4.3.3 Further Analysis of the *Chemistry* and *Animal* Programs

In the *animal* and *chemistry* programs, as processors are added, speed-up in the configurations using constraints is much less than the speed-up with no constraints. However, the programs using constraints had the best response times. This phenomenon occurs because constraints prune potential parallel branches and thus eliminate some OR parallelism. When the programs are executed without constraints, these OR-branches can be expanded on separate processors. So as more processors are added, speed-up occurs. When executing with constraints, depending on

Method	Configuration	Number of Nodes (1)	Number of Msgs.	Size of Msgs. (2)	CM Msgs. (3)	CM Msg. Time (4)
Compiled Programs	1 PSM	76	7	275	NA	NA
	2 PSMs	76	59	2913		
	4 PSMs	76	109	4715		
	6 PSMs	76	122	4522		
With CMs	1CM 1PSM	173	184	8227	176	7.96 %
	2CMs 2PSMs	168	290	13987	215	9.37 %
	3CMs 3PSMs	170	380	17695	237	10.37 %
No Constraints	1 PSM	390	7	156	NA	NA
	2 PSMs	390	45	2753		
	4 PSMs	390	150	8242		
	6 PSMs	390	348	14255		

"msg." abbreviates "message".

(1) Number of nodes in the proof tree created while finding all answers to the query.

(2) Total size of all messages sent, in bytes.

(3) Number of messages sent by or sent to CMs.

(4) Time spent by PSMs in processing CM messages, as a percent of total PSM active time.

Figure 8: Data Collected for the Chemistry Program.

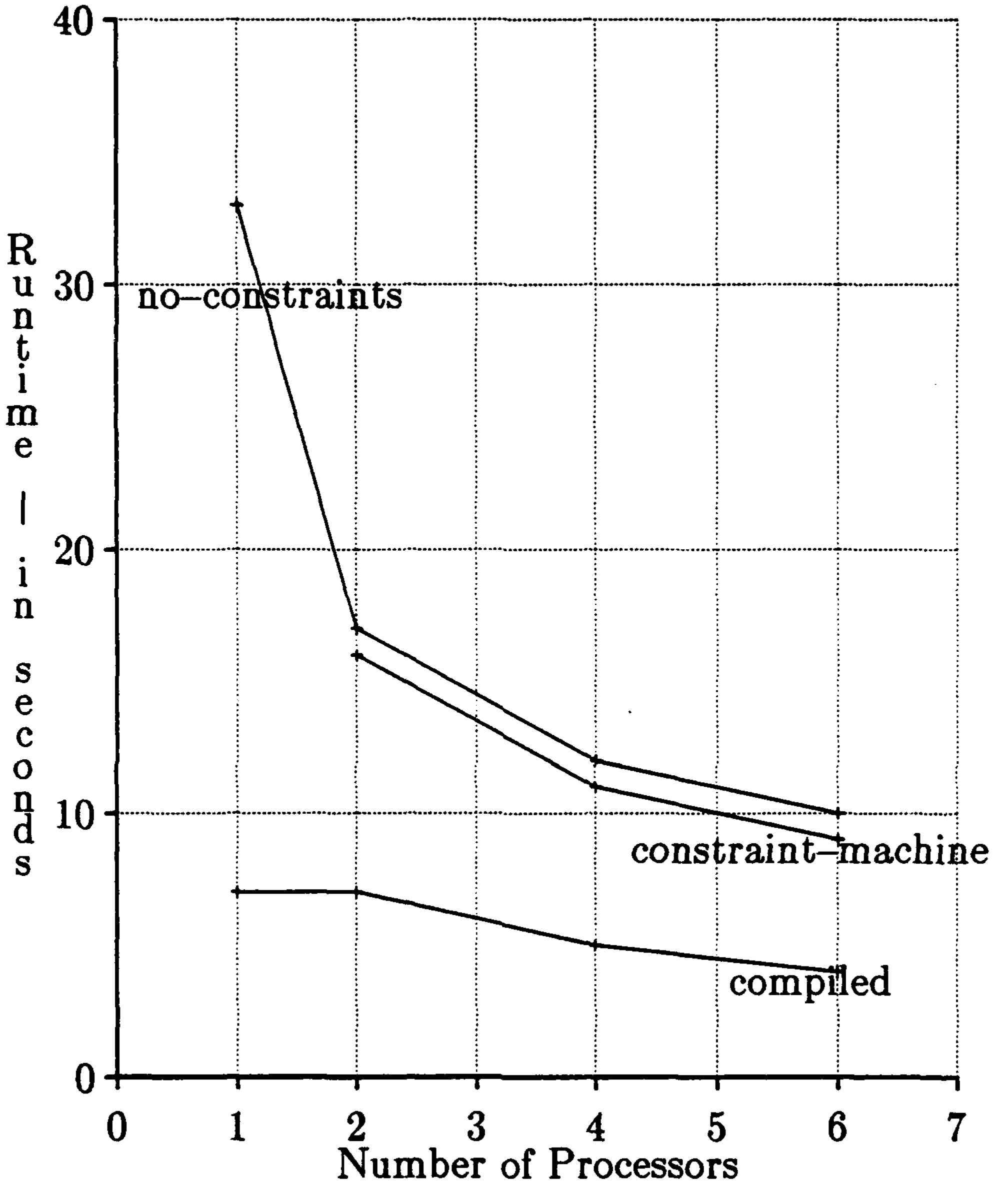


Figure 9: Runtime for the Chemistry Program

the application, some opportunities for parallelism may remain and result in speed-up as more processing elements are added.

In our experiments with the *animal* and *chemistry* programs, the response time for the compiled version run with only PSMs was consistently lower than the response time for the uncompiled program run with CM-PSM pairs. Furthermore, the response times for the CM-PSM pairs were better than the response times for the uncompiled program run only on PSMs. However, a simple example shows that in some cases, for uncompiled programs, neither the CM-PSM configuration nor the PSM-only configuration is clearly preferable.

In the following scenario, a configuration running an uncompiled program on PSMs can have the same performance or better than a configuration running the program with CMs and PSMs. Configurations with CMs assign a CM to each PSM. So given 4 processors, we could configure them as 4 PSMs or as 2 CM-PSM pairs. Consider an ideal situation in which a query's search tree has 4 identical branches. If 2 branches are pruned by the CMs, the query would have the same runtime for both configurations. Additionly, both configurations would utilize all machines. Suppose instead that one of the CMs pruned one branch and the other none. The PSM-only configuration would utilize all machines equally, while one CM would lay idle in the CM-PSM configuration. Also, the extra problem solving power could result in a better runtime using the PSM-only configuration.

The behavior of the chemistry program approximates the ideal situation. The original search tree had two branches one of which was eliminated by the CM. The remaining half of the search tree had roughly half of the opportunities for parallelism. Thus, as shown in Figure 9, the behavior of the uncompiled program run with CM-PSM pairs was similar to the behavior of the uncompiled program run on PSMs only.

4.4 Generate-and-Test Results

Generate-and-test is a common logic programming technique often used for constraint satisfaction problems. Each possible solution is generated and then checked to determine whether it satisfies the test conditions. A generate-and-test program has a clear declarative meaning and is easy to write because the generation phase is independent of the test conditions. However, such a program can potentially be inefficient. Partial evaluation [14] and coroutining [3] are existing paradigms which transform a generate-and-test program at compile time to make it more efficient. An alternative way to write generate-and-test programs is to encode the generate routine in the intensional rules and the test conditions as integrity constraints. This encoding preserves the declarative meaning of the program.

Both the compiled approach and the runtime approach can be used to apply test conditions to partially built solutions as the intensional program generates them.

Semantically compiling the constraints into the program modifies the program so that the testing phase is incorporated into the generation phase at compile time. However, to apply the test predicates to appropriate partial structures, the compiled approach requires additional runtime support. In the previous domains, the constraints were most useful when compiled into the rule for the Query predicate. In constrast, in the generate-and-test domain, the constraints must apply at each level of recursion – therefore, constraints must be compiled into the recursive rule for Generate.

Constraint tests that operate on successive data elements must be delayed until the next elements are generated. *Delayed evaluation* requires a more complex control strategy than the normal left-to-right Prolog strategy. For MuProlog, [12, 13] suggests the use of wait-predicates to achieve delayed evaluation. For the compiled programs, we used PRISM's ability to support arbitrary literal selection strategies to achieve delayed evaluation. In the CM-PSM configurations, the need for delayed evaluation is handled automatically when the CM propagates partial constraints to subqueries.

Some, but not all, generate-and-test problems can be written efficiently with constraints. The following section presents a program that can be coded efficiently and discusses our experimental data for the program. The subsequent section presents the types of constraints that cannot be coded efficently. The N-Queens example shows a specific problematic case.

4.4.1 The Zebra Program

Usually in generate-and-test programs, the generated structure is either a list or a function of a function, where the size may or may not be fixed and each element is chosen from a possible solution set. The test conditions may impose conditions on individual elements or on combinations of elements.

Consider the problem of finding "who owns the zebra". The essence of the problem is "There are five houses, each of different color, inhabited by people of different nationalities, with different pets, drinks and cigarettes. They satisfy some constraints coded as IC1 and IC2. We have to find who lives where and who owns the zebra." The following rules are the main clauses of a generate-and-test program that solves a scaled down Zebra problem involving only color and nationality:

```
Query(Houselist) ← Generate(Houselist,Colors,Nationalities,),
                Test(Houselist).
Generate([],X1,X2).
Generate([house(C,N) | Rest],Colors,Nationalities) ←
                Choose(C,Colors,RestColors),
                Choose(N,Nationalities,RestNationalities),
                Generate(Rest,RestColor,RestNationalites)).
```

The integrity constraint IC1 says that if a list of houses begins with a green house, then a red house must be the second element in the list. The integrity constraint IC2 says that no list of houses may consist of a single green house. IC1 and IC2 can be used together to impose the condition that "The green house must be immediately followed by the red house" in the solution to the problem.

```
IC1: ← Generate([house("green",N1),house(C2,N2) | R], Cs, Ns), NEQ(C2,"red").
IC2: ← Generate([house(green,N3)]).
```

If the constraints were semantically compiled into the rule for the Query predicate only, then the constraint tests would only check the top elements of the list of houses. But we want them to apply at each position in the list. When the constraints are compiled into both the rule for Generate and the rule for the Query predicate, the structure of the list is checked at each generation step.

Applying variable substitution to IC1 and IC2 produces the variable substituted forms:

```
IC1': ← Generate(X, Cs, Ns), NEQ(C2,"red"), EQ(C1, "green"),
            EQ(X, [house(C1,N1),house(C2,N2) | R]).
IC2': ← Generate(X, Cs, Ns), EQ(C3, "green"), EQ(X, [house(C3,N3)]).
```

Merging the Generate rule with IC1′ and IC2′ produces the partial constraints P1 and P2, respectively, and the semantically constrained rule R1:

```
P1: ← NEQ(C2, "red"), EQ(C1, "green"),
            EQ(Rest, [house(C1,N1),house(C2,N2) | R]).
P2: ← EQ(C3, "green"),EQ(Rest, [house(C3,N3)]).
R1: Generate([house(C,N) | Rest],Colors,Nationalities) ←
            Choose(C,Colors,RestColors),
            Choose(N,Nationalities,RestNationalities),
            (NEQ(Rest,[house(C1,N1),house(C2,N2) | R]) ∨ EQ(C2,"red") ∨ NEQ(C1,"green")),
            (NEQ(Rest, [house(C3,N3)]) ∨ NEQ(C3, "green")),
            Generate(Rest,RestColor,RestNationalites)).
```

At this point, the semantically constrained rule R1 contains disjunctions that must be eliminated by unfolding the rule. The unfolding process must preserve variables, such as C1, that do not occur in the rule outside of the disjunctions. For example, consider the first disjunction which consists of three tests and is satisfied if any one of the tests succeeds. If the variable Rest unifies with the list [house(C1,N1),house(C2,N2) | R] then the first test will fail; however, the bindings for C1 and C2 that the unification produces will be lost, and without these bindings the other two tests can not be performed. The bindings can be preserved by rewriting the first disjunction in the following equivalent form:

```
(NEQ(Rest, [house(C1,N1),house(C2,N2) | R]) ∨
            (EQ(Rest, [house(C1,N1),house(C2,N2) | R]), EQ(C2, "red")) ∨
            (EQ(Rest, [house(C1,N1),house(C2,N2) | R]), NEQ(C1, "green")))
```

Similarly, the second disjunction can be rewritten as:

```
(NEQ(Rest, [house(C3,N3)]) ∨ (EQ(Rest, [house(C3,N3)]), NEQ(C3, "green")))
```

After rewriting the disjunctions in the semantically constrained rule it can be unfolded to produce:

```
    Generate([house(C,N) | Rest],Colors,Nationalities) ←
            Choose(C,Colors,RestColors),
            Choose(N,Nationalities,RestNationalities),
            Test1(Rest),
            Test2(Rest),
            Generate(Rest,RestColor,RestNationalites)).

Test1(Rest) ← (NEQ(Rest, [house(C1,N1),house(C2,N2) | R]).
Test1([house(C1,N1),house("red",N2) | R]).
Test1([house(C1,N1),house(C2,N2) | R]) ← NEQ(C1, "green").

Test2(Rest) ← NEQ(Rest, [house(C3,N3)]).
Test2([house(C3,N3)]) ← NEQ(C3,"green").
```

Notice that Test1 must wait until the recursive call to Generate has chosen the next two elements in the list, and Test2 must wait until the next element in the list is chosen.

4.4.2 Results and Analysis for the Zebra Program

For the *chemistry* and *animal* programs, the compilation process trimmed the search space by eliminating branches at compile time. In constrast, for the *zebra* program, compilation introduced runtime tests to be performed by the PSM in order to prune the search space. As a result, the gain from performing compilation is smaller for the *zebra* program than for the *animal* and *chemistry* programs.

In further contrast to the *animal* and *chemistry* programs, the search space for the *zebra* program run with CMs was smaller than the search space for the compiled program. The CM handles partial contraint literals by generating new partial constraints for a subquery and propagating them to the subquery's children until they are evaluable. In contrast, when the partial constraint literals are added to the compiled programs, they become part of the search space that must be handled by the PSM. The CM-PSM search space was 29 percent of the original search space, whereas the compiled program space was 44 percent of the original.

Although the search space was smaller when using CMs (Figure 10, the compiled version still had faster response times than the CM version (Figure 11). This phenomenon arises because when executing with CMs, half the processors are allocated to be CMs instead of PSMs. Even after pruning, the program has much inherent parallelism. The compiled version, running on all PSMs, can fully exploit the parallelism. However, the CM version, running with half the number of PSMs, requires more processors to fully exploit the parallelism.

As Figure 11 shows, when we run the compiled program on 2 PSMs and when we run the original program with 2 PSMs and 2 CMs, we get similar response times. Also, when the compiled program is run with 4 PSMs and the original with 3 CM-PSM pairs, we get similar response times. This indicates that a more flexible processor allocation strategy which assigns more than one PSM to each CM may produce CM response times that would be better than the compiled version.

Overall, the compiled version run on 6 machines was fastest, and the configurations with constraints produced better response times than the configurations without constraints.

Method	Configuration	Number of Nodes (1)	Number of Msgs.	Size of Msgs. (2)	CM Msgs. (3)	CM Msg. Time (4)
Compiled Programs	1 PSM	273	9	387	NA	NA
	2 PSMs	273	94	7674		
	4 PSMs	273	111	9683		
	6 PSMs	273	270	20004		
With CMs	1CM 1PSM	156	169	10846	159	6.66 %
	2CMs 2PSMs	181	240	15423	199	7.57 %
	3CMs 3PSMs	198	232	14902	209	7.86 %
No Constraints	1 PSM	615	9	307	NA	NA
	2 PSMs	615	105	8185		
	4 PSMs	615	87	7016		
	6 PSMs	615	179	12588		

"msg." abbreviates "message".

(1) Number of nodes in the proof tree created while finding all answers to the query.

(2) Total size of all messages sent, in bytes.

(3) Number of messages sent by or sent to CMs.

(4) Time spent by PSMs in processing CM messages, as a percent of total PSM active time.

Figure 10: Data Collected for the Zebra Program.

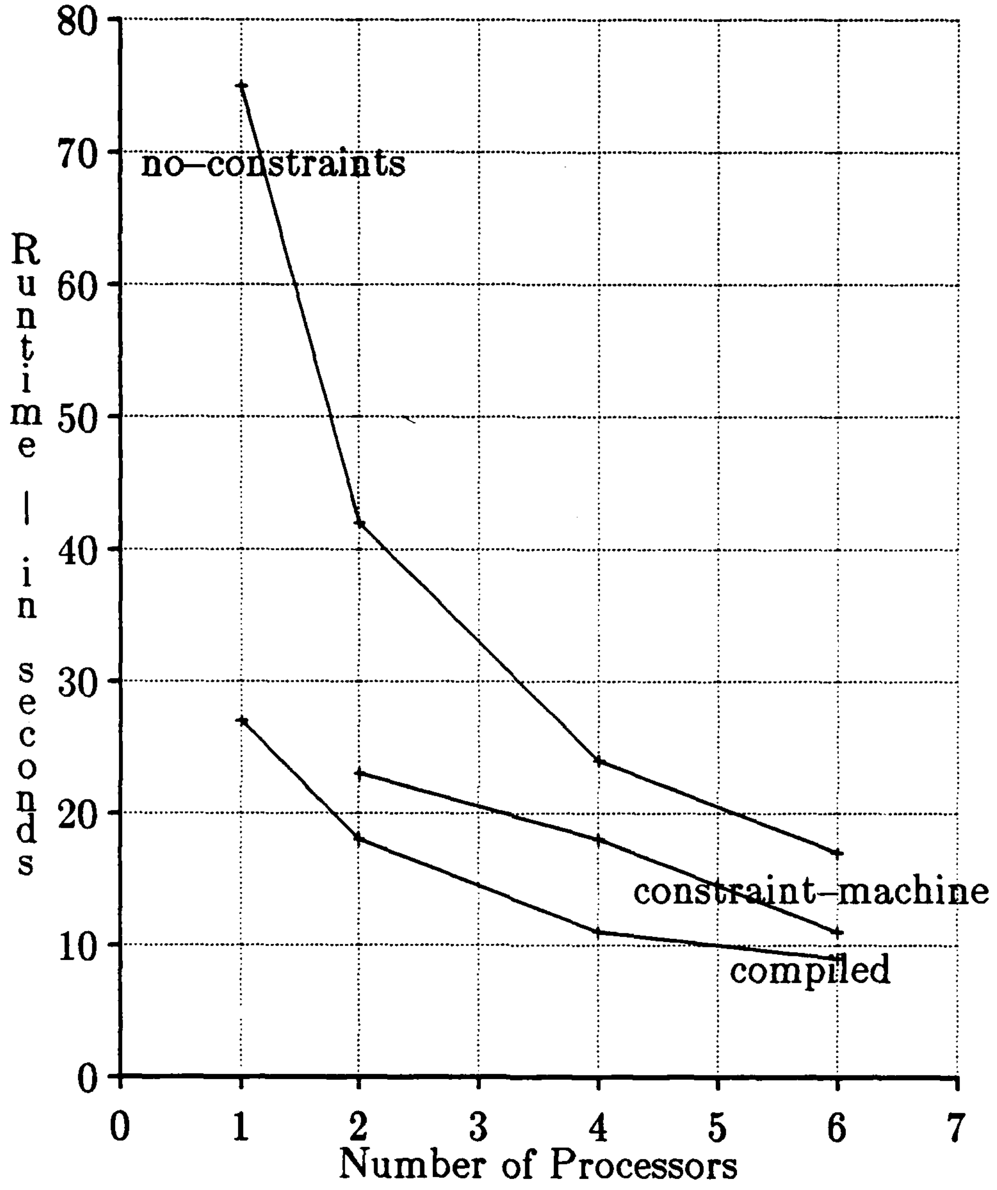


Figure 11: Runtime for the Zebra Program

4.4.3 Using Constraints With Generate-and-Test Programs

The example in the preceeding section showed a case where integrity constraints can be used to express test conditions for generate and test programs. However, it is not possible to efficiently encode all test sequences as integrity constraints. In this section the types of tests that can be efficiently encoded as integrity constraints are outlined and a problematic program is given.

In generate-and-test programs, the generating phase generates the various possible states one by one, and the testing phase rejects states which do not satisfy some conditions. Representing a state as a list of records, where a record consists of a function or a fixed-length list of functions, allows integrity constraints to be coded as restrictions on the kind of list the Generate predicate can generate. For example, Generate with its argument might look like Generate($[f_1(X_1,\ldots), f_2(X_2,\ldots),\ldots f_n(X_n,\ldots)$ | Rest]) and a test condition might restrict the domain of the first two arguments – for example, $\leftarrow$ NE(X_1, X_2).

A second issue is what predicate the constraints concern. In the example above the constraints mention the Generate predicate. However it is possible to express many of the same constraints on the Query predicate. An initial observation is that constraints on the Generate predicate apply at each postition in the list. In contrast, constraints on the Query predicate apply only at the first position of the list. This is because the generate procedure is recursively called and the constraints apply at each level of the recursion. The constraint:

$\leftarrow$ Query([f("blue","beer") | Tail]),

states that the first element of generated structure cannot have "blue" as its first argument and "beer" as its second. The constraint:

$\leftarrow$ Generate([f("blue","beer") | Tail]),

says that no list element can have "blue" as its first argument and "beer" as its second argument.

A series of test conditions are presented and constraints are provided which enforce the constraint. The encoding of the constraint using the Generate and Query predicates are contrasted.

1. A positive statement about a particular position specified from the beginning of the list.

   "The second element generated is blue"

   $\leftarrow$ Query([S,B | Tail]),NE(B,"blue").

2. A negative statement about a particular position specified from the beginning of the list.

   "The second element generated cannot be blue"

   $\leftarrow$ Query([S,"blue" | Tail]).

The previous constraints could be written using the Generate predicate only if the length of the generated structure were known in advance. The structure in the constraint would contain the proper number of elements (that is no Tail variable) and would thus apply only in the proper situation.

In addition, unless explicitly noted the following constraints can make both positive and negative statements as illustrated above.

3. A negative/positive statement about a particular position specified from the end of the list.

   "The second to last element of the list cannot be green"

   ← Generate(["green" , Last]).

The constraint can also be encoded on the Query predicate if the length of the list is known.

4. Constraints involving the inner structure of all elements generated in the list.

   "The blue house cannot be the beer house"

   ← Generate([house("blue","beer") | Tail]).

To express this constraint on the Query predicate requires O(List-size) constraints and for the length of the list to be known.

5. Constraints involving records that are a *fixed number of records* apart.

   "The blue house is immediately to the left-of the beer house"

   ← Generate([house("blue",_),house(_,R) | Tail]),NE("beer",R).
   ← Generate([house(R,_),house(_,"beer") | Tail]),NE("blue",R).

Both constraints are necessary. The first constraint allows the beer house to appear without having the blue house be to its left. The second constraint allows the blue house to appear without having the beer house to its right. This test condition would require 2*O(list-size) constraints to state using the Query predicate. In addition the list length would have to be known.

6. Constraints involving records that are an unspecified number of records apart.

   "The blue house is before the beer house"

To encode this constraint using the Generate predicate requires knowing the length of the list and requires 2*O(list-length) constraints. To encode the constraint using the query predicate requires 2*O(list-length)^2 constraints and also requires knowing the length of the list.

These examples show that the number of constraints required to express a test condition varies with the nature of the test and the predicate used to express the test. Even when a structure has a known size, and when elements in the structure have an easily describable relationship, a test which evaluates the structure to determine some property cannot be coded efficiently.

4.4.4 The N-Queens Problem

In the Zebra problem, all of the test conditions can be coded as integrity constraints. However, in the N-Queens problem, even though it is possible to code all the test conditions as integrity constraints, the resulting program is very inefficient.

Suppose we must position N queens in an N*N rectangular space so that they can not attack each other. One way to represent a positioning of queens on the board is by using lists and a function "pair". A positioning of 4 queens in an 4*4 board with all the queens on the second column is represented as [pair(4,2), pair(3,2), pair(2,2), pair(1,2)]. Consider the following program fragment to solve the problem:

```
NQueens(N,Solution) ← Generate(N,Solution) , Test(Solution).
Generate(1, [pair(1,Col)] ) ← GiveMeOneByOne(Col).
Generate(M, [pair(M,Col) | Tail) ) ← NE(M,1), SUM(N,1,M) , Generate(N,Tail),
        GiveMeOneByOne(Col).
```

This program automatically insures that the rows are unique because the rows are set by the variable M. So we only need constraints to distinguish the diagonals and columns – N-1 constraints for the columns and constraints for the diagonals. To distinguish the columns, we can have constraints such as :

```
← Generate(Rows, [pair(Row1,Col), pair(Row2,Col) | Rest]).
```

which means we can not have queens in consecutive rows and in the same column; likewise, we would need constraints like ← Generate(Rows, [pair(Row1,Col),P,pair(Row3,Col) | Rest]).

to say that we cannot have queen in the same column but two rows apart, three rows apart, up to N-1 rows apart. To distinguish diagonals, we can have a constraint such as: ← Generate(Rows, [pair(X1,Y1),pair(X2,Y2) | Rest]),
    AbsDiff(X1,X2,DX),AbsDiff(Y1,Y2,DY),EQ(DX,DY).

which ensure that queens in consecutive rows are not in the same columns. Again we would have to provide constraints for two rows apart, three rows apart up to N-1 rows apart. This approach requires that the maximum value of N be known when writing the constraints.

We conclude that using integrity constraints allows us to prune the search space to some extent for generate-and-test programs. The extent to which the search space can be reduced depends on the nature of the individual problem and its test conditions, on how many of the test conditions can be coded efficiently, and on how many integrity constraints we are willing to support. If the principle test conditions can be encoded as integrity constraints, semantic compilation can be used.

4.4.5 Alternative Approaches

Two alternative approaches, partial evaluation and co-routining, have been suggested which allow efficient execution of declaratively clean generate-and-test programs. Partial evaluation is a compile time approach which takes an existing program and transforms it into an equivalent program which executes efficiently left-to-right. Knowledge about when test predicates are ready to execute is used to unfold the original program and manipulate it so that the test predicates are inserted in the proper position. Co-routining is a run time approach. It allows generate-and-test procedures to execute in an interleaved manner. First, the generate procedure creates some structure and then the test procedure tests it.

Using the compiled approach to handling integrity constraints provides a half-way point between partial evaluation and co-routining. Compilation integrates test procedures with the generate procedures. However some program transformations are needed to yield a functioning program. In addition, the resulting program may require that test predicates delay evaluation until the arguments are sufficiently bound. Note that a delayed execution can be seen as a very simple form of co-routining. The compiled approach provides a theoretical framework which allows test conditions to be mixed with generation clauses.

4.5 Conclusions

Our experimental data shows that for three widely used knowledge base domains runtime performance for query processing can be improved by using integrity constraints. In this section, we analyze the relative merits of the runtime and compiled approaches to handling integrity constraints.

4.5.1 Runtime Approach vs No Constraints

The collected data shows that for the three domains under consideration the runtime approach performs better than using no constraints. Although this result holds for our application programs, examples show that it does not always hold. In section 4.3.3 , we described how allocating machines as CMs can hurt performance by preventing parallel processing of the search tree. Basically, when processors are allocated as CMs, they are not available for exploiting the parallelism in the program. As a result, if the CM does not trim out sufficient search space, it does not contribute to the processing of the program as much as if it were a PSM. The performance of the runtime approach versus the performance without using constraints seems to be specific to the application. If a particular query produces a proof tree that has many unproductive branches, the runtime approach may perform better. On the other hand, if most of the branches of a query's proof tree produce answers, utilizing all available processors as PSMs may yield a better response time.

In addition to the loss of PSM power, the CM configurations have a slightly higher communication overhead than the PSM-only configurations. When processors are assigned to be CM-PSM pairs rather than all PSMs, the number of exchanged messages increases. The extra messages have two sources: the communication to coordinate CM-PSM pairs and the propagation of violation information to children PSMs. Additional communication and processing occurs when a PSM sends a node to its CM partner and then sends the same node to another PSM for processing. The CM partner of the receiving PSM must send a message to the CM associated with the original PSM in order to retrieve the partial subsumption list associated with the node.

In general, the message traffic of CM-PSM configurations with few processors have a higher percentage of constraint related traffic than CM-PSM configurations with many processors. Constraint machines require a large fixed communication cost and a small variable communication cost. The fixed cost occurs because messages must be sent from the PSMs to the CMs for each node in the search tree expanded by a PSM. The variable cost involves sending constraint tree nodes between CMs. The amount of CM related messages grows slowly as more machines are utilized, and the PSM-to-PSM message traffic grows directly with the number of PSMs utilized. Thus, with larger configurations, there is more PSM-to-PSM traffic which is amortized over the fixed amount of CM traffic. For example, for the *zebra* program, 96 percent of the traffic for one CM-PSM pair is constraint related, for two pairs, 74 percent, and for three pairs, 65 percent.

The number of messages sent in the CM configuration and in the PSM-only configuration are similar because of a tradeoff between CM-PSM traffic in the CM configurations and PSM-PSM traffic in the PSM-only configurations. Overall, the overhead due to constraint machine communication is small - 8-10 percent for the *chemistry* program, 11-13 percent for the *animal* program and 7-8 percent for the generate-and-test program. These percentages are the amount of total active PSM time spent packaging, sending, receiving and processing CM messages.

4.5.2 Runtime Approach vs Compiled Approach

The experimental data shows that for the selected domains the compiled approach to using integrity constraints performs better than the runtime approach. In addition to the decrease in response time, a significant advantage of the compiled approach is the ability to utilize all available processors as PSMs and thus to maximize the amount of OR-parallel deduction.

The major cost of the compiled approach is the time to perform compilation. This time is not significant and is amortized over all the queries that the system handles after compilation. If the system processes many queries, the overhead compilation time associated with each query is very small. In contrast, if only a single query is to be processed, then using the runtime approach may be faster than performing compilation and then solving the query with the compiled approach. A minor cost of the compiled approach involves defining and maintaining "query rules" - like the rules for the predicate 'Query" that were used in the experiments.

Query rules specify a conjunction of atoms. They can be considered to define a complex view over the database. Defining complex views with query rules enables maximum pruning of the search tree under the compiled approach. When a query containing a particular conjunction of atoms occurs frequently, defining it through a query rule and compiling the rule may save time. In contrast, under the runtime approach it is not necessary to define query rules for conjunctive queries in order to achieve maximum pruning.

Running the experimental programs with the constraint machines required no changes to the programs: the programs that were run without constraint machines were also run with constraint machines. In contrast, preparing the compiled programs for execution required extra practical transformations to the programs. In order to completely automate the extra transformations, the compiler would have to be modified to detect and eliminate redundant partial constraints and to rewrite rules with disjunctions; and a final module would have to be added to the compiler to determine whether a delayed evaluation control strategy is necessary.

In addition to incurring preprocessing costs, the compiled approach can actually increase the amount of deduction performed. In the *animal* and *chemistry* programs, compilation produced a clear reduction in search space at compile time. However, in the *zebra* program, compilation introduced runtime tests which offer opportunities for but no guarantees of search space reduction. In contrast, the CM approach never introduces new deduction in the PSM because partial constraint literals are evaluated in the CM.

For the compiled approach, additional runtime effort is required to process partial constraints when queries are not known in advance. At compile time, semantic compilation can be performed on the program to attach partial constraints to individual rules. When incoming queries have a single literal in the body, the partial constraints can reduce search space as usual. However, to process a new query that is a conjunction of literals, the runtime environment of the PSM would require modification to merge the partial constraints and the other literals that result from expanding the literals in the query. For example, consider the program and integrity constraint shown below.

P ← Q.
R ← S.
IC: ← Q, S.

If we know the query ← P, R at compile time and encode it as the rule Query ← P, R, then flattening produces the new rule Query ← Q, S. The new rule body is subsumed by the literals in the integrity constraint. Hence semantic compilation will tell us at compile time that the query violates the integrity constraint. In contrast, if there is no rule that defines a Query predicate, we can merge the integrity constraint with the remaining two rules to produce the following semantically constrained rules:

P ← Q. { ← S. }
R ← S. { ← Q. }

If the query ← P, R arrives at run time we can use the semantically constrained rules to produce the following expanded query with two attached partial constraints: ← Q, S. { ← S. ← Q. }. Each partial constraint subsumes the literals in the expanded query. Hence merging the partial constraints with the expanded query at runtime tells us that the query violates an integrity constraint. The CM approach requires no additional effort to support new queries because it performs partial subsumption at runtime.

Although the problems with the compiled approach listed above may prevent it from being useful in some applications, the approach has significant advantages. First, compilation performs partial subsumption at compile time and in many cases removes significant portions of the source program before execution. Second, the approach allows all computing resources to be used as PSMs rather than tieing up half the resources as CMs. Thus, all processors are available for exploiting the parallelism in the problem. As a result, the compiled approach should be used whenever feasible.

5 Conclusions

A logic programming system augmented with constraint processing, data storage, and data manipulation capabilities forms the basis for a knowledge base system. Our experiments explore whether any benefit can be gained from using knowledge base constraints for query processing. To perform the experiments, we added constraint processing capability to an existing parallel logic programming system, PRISM.

In the problems that we investigated, constraints allow query search space to be reduced by up to eighty percent and response time to decrease by over seventy-five percent. In each problem domain, the compiled approach to using integrity constraints works better than the runtime approach. In addition, the runtime approach works better than using no constraints at all. In some cases, external factors may indicate that using the runtime approach is more desirable than the compiled approach. For example, a query asked only once takes more time to compile than the time saved by using the compiled approach. These results indicate that constraints can be used to improve query processing for the problem domains represented by the benchmark programs. Because each problem represents a well defined domain, the experiments transcend the individual problems and show that constraints are useful for many applications.

We found that several preprocessing steps were necessary to put the theoretical approaches of Kohli and Chakravarthy into practice. Both the runtime and compiled approaches required the addition of knowledge about basic number theoretic principles to the variable substitution algorithm. The compiled approach required the following additional steps: 1) the elimination of redundant partial constraints within an axiom, 2) rewriting compiled rules that contained disjunctions, and 3) the incorporation of a delayed evaluation control strategy for partial constraint literals in recursive axioms.

6 Future Work

We have identified four promising areas for further investigation: characterizations of useful classes of partial constraints; more sophisticated query transformation methods for the compiled approach; a richer language for expressing integrity constraints, and evaluation of more flexible processor allocation strategies.

If a knowledge base has a large set of integrity constraints, the set of partial constraints associated with a rule in the theory may also be large. Characterizing useful partial constraints would help select a subset of the partial constraints to be attached to the rule in semantic compilation or to be used by constraint machines.

Additional transformation methods for queries that are not known in advance can make the compiled approach more flexible. A query that consists of a conjunction of literals may have an associated set of partial constraints for each literal. To simplify the sets of partial constraints and to identify search tree nodes that violate the constraints, transformation methods must be developed to performing partial subsumption at run time.

We would like to extend our approach to be able to use integrity constraints that consist of clauses with EDB or IDB predicates in the head. This richer constraint language would allow constraints to be used with theories whose rule bodies contain negation. In addition, Generate-and-Test constraints such as those in the N-Queens problem which required 2(n-1) clauses to express the constraint could be written with a single clause in the richer language. Such integrity constraints can not be rewritten into the form $\leftarrow C_1, \ldots, C_n, E_1, \ldots, E_m$ without introducing negative EDB or IDB literals. To apply the CM approach to negated EDB or IDB literals, the CM would have to be able to handle negation and to delay deduction on IDB and EDB literals until they are sufficiently instantiated. To apply the compiled approach to constraints with negated IDB and EDB literals would involve further investigation of how to use partial constraints containing such literals.

As seen in the *zebra* program, some programs may benefit from having each CM serve more than one PSM. Future research could explore whether flexible allocation strategies or dynamic allocation of processors as CMs and PSMs is appropriate for certain applications.

Acknowledgements

We wish to express our appreciation to the National Science Foundation for their support of our work under grant number IRI-86-09170 and the Air Force Office of Scientific Research under grant number AFOSR-88-0152. We wish to thank Chitta Baral, Jorge Lobo, Zheng Lin, Shiva Azadegan, and Arcot Rajasekar for their help and participation in the development of the ideas presented in the paper.

APPENDIX

A Variable Substitution Algorithm

Variable Substitution Algorithm

```
For each occurrence $t$ of a term in the integrity constraint, do
    If $t$ occurs in an atom whose predicate is defined via facts or rules in the program, and
    $t$ is (a string, or a number, or a term $f(t_1, \ldots, t_n)$, or a variable that has occurred
    in another atom whose predicate is defined via facts or rules in the program), then
        replace $t$ with a new variable $y$, and
        create an equality atom $(y = t)$, and
        append the equality atom to the end of the list of literals in the constraint. $\triangleleft$
```

The following is an example of an integrity constraint and the corresponding output formula produced by the CMC, respectively:

$$\leftarrow p(x), q(f(x)), LESS(x, 3)$$

$$\leftarrow p(x0), q(x2), LESS(x0, 3), EQ(x1, x0), EQ(x2, f(x1))$$

where LESS represents $<$ and EQ represents $=$.

B Partial Subsumption Algorithm

Executing the test for partial subsumption of a conjunction of literals by a constraint efficiently is not as straightforward as it may seem. Consider the following situation. Suppose the constraint under consideration, C, is $\leftarrow P(x,y), Q(z), EQ(x,z)$. If the conjunction, A, contains the literals $P(u,v), S, T$, then since $P(x,y)$ in C subsumes $P(u,v)$ in A, the partial constraint that results is $\leftarrow Q(z), EQ(u,z)$. If, however, A contains the literals $P(u,v), P(v,w), S, T$, then C can partially subsume A in two different ways: one where $P(x,y)$ subsumes $P(u,v)$ and the other where $P(x,y)$ subsumes $P(v,w)$. The two partial constraints that result are $\leftarrow Q(z), EQ(u,z)$ and $\leftarrow Q(z), EQ(v,z)$. Furthermore, if $Q(z)$ also subsumes more than one literal in A, e.g. A being $P(u,v), P(v,w), Q(a), Q(b), Q(c)$, then from each of the above two partial constraints, three new ones will result.

The list of partial constraints obtained from a constraint C and a conjunction of literals A can be represented in a structure called a *subsumption tree*. Each node in this tree consists of two sets of literals, CC and B. The Partial Subsumption Algorithm is used to construct the tree.

(Partial) Subsumption Algorithm

0. Create a root node in which CC is the set of literals in the constraint C, and B is the set of literals in the conjunction A. Mark this node as LIVE.

1. Choose a node marked LIVE, and mark it USED. Let its CC part be P = P1,...,Pn and its B part be R = R1,...,Rm. Let i and j both be 1.

2. If Pi subsumes Rj with substitution θ, then go to step 3. Otherwise, go to step 6.

3. Apply θ to all the literals in the set P - Pi to produce the set P′.

4. Remove from P′ all the evaluable literals that evaluate to *true*. If this makes P′ empty, then go to step 11. Otherwise, if any of the evaluable literals in P′ evaluates to *false*, then go to step 6.

5. Create a child node for this node. The CC part of the new node is P′ and its B part is the set R - Rj. If the B part is empty, then mark the new node USED, otherwise, mark it LIVE.

6. Increment j. If the result is m+1, then do the next step. Otherwise, go back to step 2.

7. (All of Rj have been tested) If no new node is created in steps 3-6, then increment i and do the next step. Otherwise, go to step 9.

8. If i becomes n+1, then do the next step. Otherwise, set j to 1 and go back to step 2.

9. If there is still any node marked LIVE then go to step 1. Otherwise, proceed to the next step.

10. If no new node was added to the tree in all the above steps, then terminate the procedure with a result that C does not (fully or partially) subsume A. Otherwise, the CC part of each of the leaf nodes represents a partial constraint. Collect all such partial constraints to form a list L. Terminate the procedure with the result that C partially subsumes A and produces the list L.

11. (From step 4) Terminate the procedure with the result that C fully subsumes A, i.e. a violation has been detected. ◁

C Program Listings

C.1 Animals Program: Semantic Hierarchy Domain

```
QUERY1(x,y) <- eats(x,y).
QUERY2(x) <- (swimm(x),flys(x)).

swimm(x)<-is_fish(x).
swimm(x)<-is_mammal(x),lives_in_sea(x).
swimm(x)<-is_bird(x),has_webbed_feet(x).
lives_in_sea(x)<-is_fish(x).
lives_in_sea(x)<-itype(x,"whale").
flys(x)<-is_bird(x).
flys(x)<-is_mammal(x),has_wings(x).
has_wings(x)<-itype(x,"bat").
has_wings(x)<-has_feathers(x).
has_webbed_feet(x)<-itype(x,"duck").
has_feathers(x)<-is_bird(x).
bears_young_live(x)<-is_mammal(x).
has_eggs(x)<-is_fish(x).
has_eggs(x)<-is_bird(x).
furry(x)<-is_mammal(x),lives_on_land(x).
lives_on_land(x)<-is_mammal(x).
lives_on_land(x)<-is_bird(x).
has_four_limbs(x)<-lives_on_land(x),is_mammal(x).
carnivore(x)<-has_fangs(x),is_mammal(x).
carnivore(x)<-is_fish(x),large_mouth(x).
herbivore(x)<-is_fish(x),small_mouth(x).
herbivore(x)<-has_molars(x),is_mammal(x).
has_fangs(x)<-is_cat(x).
has_molars(x)<-is_cow(x).
has_fins(x)<-lives_in_sea(x).
large_mouth(x)<-itype(x,"shark").
small_mouth(x)<-itype(x,"guppy").

eats(x,y)<-lives_together(x,y),carnivore(x),herbivore(y),sm(y,x).
eats(x,y)<-is_cat(x),is_bird(y).

lives_together(x,y)<-flys(x),flys(y).
lives_together(x,y)<-lives_on_land(x),lives_on_land(y).
lives_together(x,y)<-lives_in_sea(x),lives_in_sea(y).


fish(x)<-subtype(x,"fish").
is_fish(x)<-itype(x,y),fish(y).
animal(x)<-subtype(x,"animal").
is_animal(x)<-itype(x,y),animal(y).
mammal(x)<-subtype(x,"mammal").
is_mammal(x)<-itype(x,y),mammal(y).
bird(x)<-subtype(x,"bird").
is_bird(x)<-itype(x,y),bird(y).
cat(x)<-subtype(x,"cat").
is_cat(x)<-itype(x,y),cat(y).
cow(x)<-subtype(x,"cow").
```

```
is_cow(x)<-itype(x,"cow"),cow(x).

subtype(x,x).
subtype(x,y)<-(constant(y),isubtype(z,y),subtype(x,z)).

constant("duck").
constant("bird").
constant("whale").
constant("cod").
constant("cat").
constant("shark").
constant("guppy").
constant("bat").
constant("cow").
constant("animal").
constant("mammal").
constant("fish").

itype("daffy","duck").
itype("tweety","bird").
itype("willie","whale").
itype("homer","cod").
itype("morris","cat").
itype("sam","shark").
itype("fred","guppy").
itype("dracula","bat").
itype("elsie","cow").


isubtype("duck","bird").
isubtype("bat","mammal").
isubtype("cat","mammal").
isubtype("fish","animal").
isubtype("bird","animal").
isubtype("mammal","animal").
isubtype("cod","fish").
isubtype("whale","mammal").
isubtype("cow","mammal").
isubtype("shark","fish").
isubtype("guppy","fish").


smaller(x,y)<-itype(x,"cat"),itype(y,"cow").
smaller(x,y)<-itype(x,"bat"),itype(y,"cat").
smaller(x,y)<-itype(x,"cat"),itype(y,"shark").
smaller(x,y)<-itype(x,"cod"),itype(y,"shark").
smaller(x,y)<-itype(x,"shark"),itype(y,"whale").
smaller(x,y)<-itype(x,"guppy"),itype(y,"cod").
sm(x,y)<-smaller(x,y).
sm(x,y)<-(smaller(x,z),sm(z,y)).


INTEGRITY CONSTRAINTS:

<-has_molars(x),has_fangs(x).
<-carnivore(x),herbivore(x).
<-is_fish(x),is_mammal(x).
<-is_mammal(x),is_bird(x).
<-is_bird(x),lives_in_sea(x).
<-is_bird(x),is_fish(x).
<-is_cat(x),is_fish(x).
<-is_cat(x),is_bird(x).
<-lives_on_land(x),lives_in_sea(x).
<-lives_on_land(x),fish(x).
```

C.2 Chemistry Program: Data Clustering Domain

```
% Database of physical properties of metals
% Relation schema:
%   metal(atomic_number,symbol,name,atomic_weight,density,melting_point)
% Units: atomic_weight, *100
%  density, (g/cm^3 at 20 degrees Celsius)*100
%  melting_point, (degrees Celsius)*100
%
% Note:  Wherever the periodic table has an occurrence of a null value,
%  we replace this by 0.
%  Wherever the periodic table has an occurrence of a negative number,
%  we replace this by 1.
%
query1(x)<-(in_ptable(x),is_metal(x),melting_point(m,x),
  LESS(100000,m),LESS(m,180000)).

in_ptable(x)<-in_group(g,x).

in_group("IA",x)<-member(n,[1 3 11 19 37 55 87]),
  atomic_number(n,x),is_element(x).
in_group("IB",x)<-member(n,[29 47 79]),
  atomic_number(n,x),is_element(x).
```

```
is_metal(x)<-(atomic_number(n,x),
  range(n,[[2 5] [10 14] [18 32] [36 51] [54 84] [86 104]])).

range(n,[[k1 k2]|t])<-LESS(k1,n),LESS(n,k2).
range(n,[h|t])<-range(n,t).

atomic_number(number,element(number,symbol,name,weight,density,meltpt)).
melting_point(meltpt,element(number,symbol,name,weight,density,meltpt)).

member(h,[h|t]).
member(x,[h|t])<-member(x,t).

is_element(element(3,"Li","lithium",694,053,18060)).
is_element(element(11,"Na","sodium",2298,097,9780)).
is_element(element(19,"K","potassium",3910,087,6370)).
is_element(element(37,"Rb","rubidium",8546,153,3900)).
is_element(element(55,",Cs","cesium",13290,1279876,2880)).
is_element(element(87,"Fr","francium",22300,0,2700)).

is_element(element(29,"Cu","copper",6354,892,108450)).
is_element(element(47,"Ag","silver",10786,1050,96193)).
is_element(element(79,"Au","gold",19696,1930,106443)).

INTEGRITY CONSTRAINTS:

<-in_group("IA",x),is_metal(x),
  melting_point(m,x),LESS(u,v),LE(m,u),LE(v,2700).
<-in_group("IA",x),is_metal(x),
  melting_point(m,x),LESS(u,v),LE(18060,u),LE(v,m).
<-in_group("IB",x),is_metal(x),
  melting_point(m,x),LESS(u,v),LE(m,u),LE(v,96193).
<-in_group("IB",x),is_metal(x),
  melting_point(m,x),LESS(u,v),LE(108450,u),LE(v,m).
```

C.3 Zebra Program: Generate and Test Domain

```
QUERY(x) <- (houses(Houselist), generate(Houselist,
                       ["red" "green" "blue"  ] ,
                       ["english" "spaniard" "japaneese"]),
                              does_not_conflict(Houselist),
                              print_houses(Houselist)).

houses([house(C1,N1)house(C2,N2)house(C3,N3)]).
```

```
generate([],X1,X2).
generate([house(Y1,Y2)|Rest],Col,Nat) <-
(choose(Y1,Col,Rcol),choose(Y2,Nat,Rnat),
generate(Rest,Rcol,Rnat)).

choose(A,[A|List],List).
choose(A,[B|List],[B|List1]) <- choose(A,List,List1).

member(X, [X|Y]).
member(X, [A|B]) <- member(X,B).

right_of(A,B, [B A | S1] ).
right_of(A,B, [X | Y] ) <-  right_of(A,B,Y).

next_to(A,B, [A B|S2]).
next_to(A,B, [B A|S3]).
next_to(A,B, [X | Y] ) <- next_to(A,B,Y).

does_not_conflict(Houselist)<-
(member(house("green","spaniard"),Houselist),
member(house("blue","english"),Houselist),
member(house("red","japaneese"),Houselist),
    right_of( house("red",X4) , house("green",X6) , Houselist) ,
next_to( house("green",X9) , house("blue",X11) , Houselist)).

print_houses([A|B]) <- (WRITE(A),print_houses(B)).
print_houses([]).

INTEGRITY CONSTRAINTS:

<- generate(CONS(house(col,"english") , R),A,S),NE(col,"blue").
<- generate(CONS(house(col,"spaniard") , R),A,S),NE(col,"green").
<- generate(CONS(house(col,"japaneese") , R),A,S),NE(col,"red").

<- generate(CONS(house("green",X),CONS(house(Y,Z) , R)),A,S),
     NE(Y,"red").

<- generate(CONS(house(Y,X),CONS(house("red",Z) , R)),A,S),
 NE(Y,"green").

<- generate(CONS(house(C,B),
CONS(house("green",Z),
CONS(house(Y,W) , R))),A,S),
     NE(C,"blue"),NE(Y,"blue").
```

```
<- generate(CONS(house(C,B),
CONS(house("blue",Z),
CONS(house(Y,W) , R))),A,S),
     NE(C,"green"),NE(Y,"green").
```

References

[1] U. Chakravarthy. *Semantic Query Optimization in Deductive Databases.* PhD thesis, University of Maryland, Department of Computer Science, College Park, 1985.

[2] U. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan-Kaufmann, 1987.

[3] K. Clark, F. McCabe, and S. Gregory. Ic-prolog language features. In K. Clark and S-A. S-A. Tarnlund, editors, *Logic Programming,* pages 253–266. Academic Press, New York, 1982.

[4] M. Giuliano, M. Kohli, J. Minker, and I. Durand. Prism: A testbed for parallel control. In L. Kanal and V. Kumar, editors, *Parallel Algorithms for Machine Intelligence,* page (to appear).

[5] J. Grant and J. Minker. Integrity constraints in knowledge based systems. Technical Report UMIACS-TR-89-39 CS-TR-2223, University of Maryland, College Park, Dept. of Computer Science, 1989.

[6] S. Kasif, M. Kohli, and J. Minker. Prism: A parallel inference system for problem solving. Technical Report TR-1243, University of Maryland, College Park, Dept. of Computer Science, 1983.

[7] M. Kohli. *Controlling the Execution of Logic Programs.* PhD thesis, University of Maryland, Department of Computer Science, College Park, 1987.

[8] M. Kohli, M. Giuliano, and J. Minker. An overview of the prism project. *Computer Architecture News,* 15(1):35–42, March 1987.

[9] M. Kohli and J. Minker. Control in logic programs using integrity constraints. In *Proceedings of Logic Programming Workshop,* Universidade Nova de Lisboa, Portugal, June 1983.

[10] R. Kowalski. *Logic for Problem Solving.* Elsevier North Holland Inc, New York, New York, 1979.

[11] J. Lobo and J. Minker. A metainterpreter to semantically optimize queries in deductive databases. In L. Kerschberg, editor, *Proc. Workshop on Expert Database Systems,* pages 387–420, Tysons Corner, Virginia, April 1988.

[12] L. Naish. The mu-prolog 3.2 reference manual. Technical report, Dept. of Computer Science, University of Melbourne, 1985.

[13] L. Naish. *Negation and Control in Prolog.* PhD thesis, University of Melbourne, Dept of Computer Science, College Park, 1985.

[14] H. Nakagawa. Prolog program transformations and tree manipulation algorithms. *Journal of Logic Programming,* 2(2), 1985.

[15] R. Reiter. Deductive question answering on relational databases. In H. Gallaire J. Minker, editor, *Logic and Data Bases*, pages 149–177. Plenum Press, New York, 1978.

[16] Unknown. Standard ref. for zebra program.